System Administration Toolkit: Standardizing your UNIX command-line tools

Skill Level: Intermediate

Martin Brown (mc@mcslp.com)
Freelance Writer
Consultant

22 Aug 2006

Examine methods for standardizing your interface to simplify movements between different UNIX® systems. If you manage multiple UNIX systems, particularly in a heterogeneous environment, then the hardest task can be switching between the different environments and performing the different tasks while having to consider all of the differences between the systems. This article does not cover specific differences, but you'll look at ways that can provide compatible layers, or wrappers, to support a consistent environment.

About this series

The typical UNIX® administrator has a key range of utilities, tricks, and systems he or she uses regularly to aid in the process of administration. There are key utilities, command-line chains, and scripts that are used to simplify different processes. Some of these tools come with the operating system, but a majority of the tricks come through years of experience and a desire to ease the system administrator's life. The focus of this series is on getting the most from the available tools across a range of different UNIX environments, including methods of simplifying administration in a heterogeneous environment.

Differences and issues

If you use a variety of UNIX hosts, particularly if each host supports a different UNIX flavor (Berkeley Software Distribution (BSD), UNIX System Release 4 (VSVR4), and for forth), or version, then you probably find that you spend a lot of your time

checking and determining the host you are on so that you can account for changes in the way the systems operate.

For example, the `ps` command requires different command-line options between BSD and SVR4-based UNIX hosts to get, on the whole, the same information (see System Administration Toolkit: Process administration tricks for more details). There are also more extensive differences between platforms. Sometimes it is the name of the command that has changed; Linux® provides the `adduser` command, while Solaris provides the `useradd` command.

There are a number of avenues available to you in terms of standardization:

- You could choose to standardize on your main platform, for example, Solaris, and provide wrappers around the equivalent commands on other platforms to match the Solaris standard.
- You could choose to standardize on a mixture of commands that provide the best combination for the tasks that you use, picking and choosing the commands that you like and building a wrapper around the ones that don't exist on specific platforms.
- You could create your own suite of scripts that do specific tasks, including your own replacements for common tools like `ls`, `ps`, and others so that they generate the information you want. This is dangerous only because it means you might never use the originals, which could cause potential problems in the event that your scripts are unavailable.

How you actually achieve the wrapper around the various commands to provide a compatible, or unique, layer depends on whether you are trying to simply provide a recognized name to an identical, alternative, command, or whether you need to build a wrapper around one or more commands to achieve the equivalent result. There are three potential solutions:

- Aliases -- This solution is only supported in some shells -- aliases provide a simple method for expanding a given string into a specific command.
- Shell functions -- This solution is supported by most modern shells -- shell functions enable you to create more complex sequences but, because they operate as a built-in, they can be more practical when the differences are comparatively minor.
- Shell scripts -- When the wrapper you are building is particularly complex, a much better solution is to use a shell script that you can call in place of the original command. With a shell script, you can get more creative with the replacement, even providing a completely shell script-driven

alternative to another command.

Let's look at each potential solution and some sample commands that can be emulated through this method.

Using aliases

Aliases, supported in the Korn (ksh), Bourne-Again SHell (bash), TENEX C shell (tcsh), and Z shell (zsh) shells, provide probably the most straightforward method when you want to set specific options to a command, while still supporting further options. The alias is exactly as the name suggests, you can alias one command for another, or you can provide an alias for the same command with additional options. The alias is expanded from what you type to its expansion.

For example, a common alias is `ll`, which invokes the equivalent of `ls -l` (`ll` is often referred to as *long listing*). Whenever a user types `ll`, it is directly replaced with the expansion, hence: `$ ll a*` expands before execution to: `$ ls -l a*`.

Command-line options continue to work as well, in other words, `$ ll -a` expands to: `$ ls -l -a`.

You can also alias an existing command; I have the `-F` option added to all `ls` commands, such that: `$ ls` expands to: `$ ls -F`.

To set an alias, use the built-in shell alias statement, specifying the required expansion in quotes. For example, to set up the `ll` expansion detailed earlier, use: `$ alias ll='ls -l'`.

Aliases are most useful in those situations where you want to use the `base` command and easily specify additional options, while still allowing for platform-specific options to be set.

A good example is the `ps` command, which differs on SVR4 and BSD-based UNIX hosts. In the first article in this series, see System Administration Toolkit: Process administration tricks -- this article explains how to use the options of `ps` to get similar listings. You can use this with aliases without affecting your ability to specify additional options. For example, on BSD, you would specify an alias like Listing 1.

Listing 1. Specifying an alias on BSD

```
$ alias ps='ps -o pid,ppid,command'
```

While on an SVR4 host, you would create the alias like Listing 2.

Listing 2. Specifying an alias on SVR4

```
$ alias ps='ps -opid,ppid,cmd
```

Now, within the limitations of the way the two systems operate with respect to `ps`, you have a standard output from `ps`. As before, you can continue to add further options; for example, request all processes on either platform with the alias installed, as in the case of adding the `-A` option. It produces something like Listing 3 on BSD (in this case, Mac OS X).

Listing 3. Using the -A option on BSD

```
$ ps -A
  PID  PPID COMMAND
    1     0 /sbin/launchd
   23     1 /sbin/dynamic_pager -F /private/var/vm/swapfile
   27     1 kextd
   32     1 /usr/sbin/KernelEventAgent
   33     1 /usr/sbin/mDNSResponder -launchdaemon
   34     1 /usr/sbin/netinfod -s local
   35     1 /usr/sbin/syslogd
   36     1 /usr/sbin/cron
   37     1 /usr/sbin/configd
   38     1 /usr/sbin/coreaudiod
   39     1 /usr/sbin/diskarbitrationd
...
```

An SVR4 system (Gentoo Linux host) displays the same columns, as shown in Listing 4.

Listing 4. Using the -A option on SVR4

```
$ ps -A
  PID  PPID CMD
    1     0 init [3]
    2     1 [migration/0]
    3     1 [ksoftirqd/0]
    4     1 [watchdog/0]
    5     1 [migration/1]
    6     1 [ksoftirqd/1]
    7     1 [watchdog/1]
    8     1 [events/0]
    9     1 [events/1]
   10     1 [khelper]
   11     1 [kthread]
   14    11 [kblockd/0]
   15    11 [kblockd/1]
   16    11 [kacpid]
...
```

Another option, which more or less mirrors the scripting and function solutions given

elsewhere in this article, is to create aliases for specific outputs of given commands that employ the same method to provide an identical formatted output. Staying with the `ps` example, you could create the alias `ps-all` to output a list of all processes, setting an appropriate expansion for each platform, as desired.

The best place to set these aliases is in your shell initialization script that is executed during login, such as .ksh, .profile, or .bashrc. You can perform the same system checks here to verify which aliases to enable. If you want to provide a global solution that works for all users, then place the alias definitions into a publicly available file, for example, in /etc or /usr/local, and set the user initialization scripts to source the alias definitions.

The alias system works best where you want to set the command-line options on an individual command, although they can be used to expand a given command into a set of multiple commands, or pipes. This breaks the ability to specify additional arguments to any command but the last in the expansion. For handling that kind of wrapper, an inline function within your shell might be a better bet.

Using inline shell functions

Most shells support functions, which are basically mini scripts into which you can place commands and other shell-scripted elements to perform a specific task. Because they are functions within the main shell definition, they are quick and easy to use, while still supporting many of the same functionality as a full-blown shell script, such as command-line arguments.

The support for command-line arguments is key to supporting certain commands and combinations where an alias would be unable to work. For example, the `killall` command, at its most basic level, kills all commands that match a particular string. It is not available on all platforms but, once you find out about it, you'll want to use it within other environments.

On Solaris, the `killall` command exists, but it is used as part of the shutdown process to kill all processes. Imagine accidentally calling the `killall` command on a Solaris host to shutdown all Apache processes, only to find you've effectively shutdown the system!

Providing a replacement -- use the same name, or use a different name, on all hosts -- achieves the desired result of killing processes by their name, and it eliminates unwanted and potentially costly mistakes, while extending the functionality of the systems that don't naturally support the option.

The key component to the command is the ability to identify running processes,

extract the ones matching a given string, and use the `kill` command to send the `KILL` signal to each matching process. On the command line, you can achieve the equivalent through a series of pipes (using the `KILL` signal), as shown in Listing 5.

Listing 5. Providing a replacement for the killall command

```
$ ps -ef|grep gcc|awk '{ print $2; }'|xargs kill -9
```

The key elements to the command are the strings provided to grep (gcc in this case) and the column from the output of `ps` that holds the process ID that you need. The above example is valid for a Solaris host and most SVR4 UNIX variants.

An alias will not work in this example, because the information you want to be able to insert into the command is not at the end; aliases work through a method of expansion. However, an inline shell function is perfect.

Within shells supporting the Bourne syntax (bash and zsh), you can define a function using the following syntax shown in Listing 6.

Listing 6. Defining a function

```
function NAME()
{
# do stuff here
}
```

Arguments to the function when it is called are available as $1, $2, and so forth just within a typical shell script. Therefore, you can define a function that performs the same string-based signaling as `killall` (see Listing 7).

Listing 7. Defining a function that performs the same signaling as killall

```
function killall()
{
    ps -ef|grep $1|awk '{ print $2; }'|xargs kill -9
}
```

Note that the $2 within the awk portion of the function will not be expanded, because you have used single quotes for the awk script definition, which prevents expansion and, in this case, picks out the second column.

As with aliases, the best place to specify shell functions is within the initialization script for your shell. The limitation of functions is that they rely on the ability of support being available within the shell, which is not always possible, or available.

Although you can make an inline shell function as long you like, there are many occasions when the shell function is less than ideal. For example, in very long sequences where you are simulating a more complex command or where you are providing a wrapper around a command you need to parse options and provide the localized equivalent, the inline function is less useful. A shell script might be more suitable here.

Using scripts

The easiest and most compatible way to build a consistent environment is to create shell scripts that can be used as a wrapper around the real commands, which account for the various options and settings that you want to support.

For example, the `useradd` and `adduser` command supports the same single-letter command-line options when setting parameters, such as user ID or group membership, so the Linux: `$ adduser -u 1000 -G sales,marketing mcbrown` is equivalent to the Solaris: `$ useradd -u 1000 -G sales,marketing mcbrown`.

However, the Linux version also supports extended command options, for example, `--uid` and `--groups` are equivalent to the above command-line options. These are not supported on Solaris but, if you create a shell script with the name `adduser`, you can simulate the Linux version and then run the real Solaris `useradd` command with appropriate options.

Listing 8 is a sample shell script that acts as a wrapper around either the `adduser` or `useradd` command.

Listing 8. Sample shell script that acts as a wrapper

```
#!/bin/bash
# -*- shell-script -*-


for i in $*
do
  case $i in
      --uid|-u) OPT_UID=$2; shift 2;;
      --groups|-G) OPT_GROUPS=$2; shift 2;;
      --gid|-g) OPT_GROUP=$2; shift 2;;
      --home-dir|-d) OPT_HOMEDIR=$2; shift 2;;
      --shell|-s) OPT_SHELL=$2;shift 2;;
      --non-unique|-o) OPT_NONUNIQUE=1;shift 2;;
      --comment|-c) OPT_COMMENT=$2;shift 2;;
  esac
done

OPTS=""
```

```
if [ -n "$OPT_$HOMEDIR" ]
then
    OPTS="$OPTS -d $OPT_HOMEDIR"
fi

if [ -n "$GROUP" ]
then
    OPTS="$OPTS -g $OPT_GROUP"
fi

if [ -n "$OPT_GROUPS" ]
then
    OPTS="$OPTS -G $OPT_GROUPS"
fi

if [ -n "$OPT_SHELL" ]
then
    OPTS="$OPTS -s $OPT_SHELL"
fi

if [ -n "$OPT_UID" ]
then
    OPTS="$OPTS -u $OPT_UID"
fi

if [ -n "$OPT_COMMENT" ]
then
    OPTS="$OPTS -c \"$OPT_COMMENT\""
fi

if [ -n "$OPT_NOUNIQUE" ]
then
    OPTS="$OPTS -o"
fi

CMD=adduser

UNAME=`uname`

case $UNAME in
    Solaris) CMD=useradd;break;;
esac

$CMD $OPTS $*
```

The key to the script is the `foreach` loop that works through the command-line arguments supplied (and available in `$*`). For each option, the `case` statement then attempts to identify the option, using either the short or long format and setting a variable. The command-line switch is `$1`. If the option is normally followed by a value (for example, the user ID), then you can access this as `$2`, which you use to assign the value to a variable.

When an option has been identified, the `shift` statement removes one (or more if a number is specified) from the `$*` variable list so that the command-line arguments that have already been identified are no longer in the `$*` variable in the next iteration of the loop.

With the potential arguments identified and extracted, all you need to do is build the new options to supply to the command that will ultimately be used. Since useradd/adduser both support the short form of the arguments, a new command option string can be built on this basis. This is achieved by checking whether the corresponding variable has been set and adding the option to the command line. Note the use of double quotes, which ensures that quoted arguments in the original are retained and identified properly.

Installed on a platform supporting either original command, you can now add a user specifying the options you want, including mixing and matching arguments (see Listing 9).

Listing 9. Adding a user

```
$ adduser.sh --homedir /etc -g wheel --shell /bin/bash -c "New user" mcbrown
```

The same basic principles can be used to build wrappers around other commands, even changing argument names and options, or providing equivalent expressions.

If you want to install this with the original name -- for example, `adduser` -- and place it into a directory (for example, /usr/local/compat), you must ensure that that directory appears before the real command directory in the `PATH`. Here's an example if you place your compatibility scripts in the /usr/local/compat directory: `$ PATH=/usr/local/compat:$PATH`.

Using a single source

Whether you are using multiple scripts or a single configuration script/aliases to support your unified environment, you will probably want to use a single suite of scripts to support your system. So, setting up a new system to use your standardized scripts, whether they are standalone or setup shell functions and aliases, is as simple as copying them to the new system.

You can use a single source by using a combination of command-line tools and shell flow control (such as `if` or `case`) to select the various options to use. Two tools are useful here: one that identifies your host (such as hostname or uname) and one that identifies your platform (uname).

The default output from uname is the base operating system name, such as Linux or Solaris. For example, you can use it in combination with a `case` statement to choose the right aliases, as per the `ps` examples in the previous section, like Listing 10.

Listing 10. Output for uname

```
UNAME='uname'
case "$UNAME" in
FreeBSD|NetBSD|Darwin)
    alias ps='ps -o pid,ppid,command'
    break
    ;;
Solaris|Linux)
    alias ps='ps -o pid,ppid,cmd'
    break
    ;;
esac
```

The same basic process can be used in a script to choose a particular sequence.

When using inline shell functions, it is generally easier to select the right function definition using a wrapper like this, rather than making the decision each time the function is used, because it will be more efficient that way.

Summary

Normalizing your environment goes a long way to simplifying administration. It relieves you of the burden of working out which system you are on and which command and/or set of options is most appropriate to get the information, or perform the operation, that you need. Choosing the right system for each command depends entirely on the command and what you are trying to achieve.

On single commands where you want to invoke command-line options, it is best to use the alias system. Inline functions are best for more complex operations and sequences that you want to easily embed into your current script environment, while full-blown, individual scripts are best for the complicated, multi-step operations, or where you want to provide support for a command, or option, without making changes to your shell environment.

Despite the obvious advantages, it is important to remember that shielding yourself too much from the underlying system might leave you in an unprepared state should a failure occur, and you don't have access to the scripts -- you should seek to extend and augment, rather than replace.

Resources

Learn

- System Administration Toolkit: Check out other parts in this series.
- Working in the bash shell (developerWorks, May 2006): This tutorial provides a guide for using the Bourne Again Shell for all your work, including methods for customizing and extending your environment.
- Making UNIX and Linux work together (developerWorks, April 2006): This article is a guide to getting traditional UNIX distributions and Linux working together.
- Bash: Bash is an alternative shell to the standard Bourne shell with similar syntax, but an expanded range of features, including aliasing, job control, and auto-completion of file and directory names.
- AIX® and UNIX articles: Check out other articles written by Martin Brown.
- Search the AIX and UNIX library by topic:
  - System administration
  - Application development
  - Performance
  - Porting
  - Security
  - Tips
  - Tools and utilities
  - Java technology
  - Linux
  - Open source
- AIX and UNIX: The AIX and UNIX developerWorks zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- New to AIX and UNIX: Visit the New to AIX and UNIX page to learn more about AIX and UNIX.
- AIX 5L™ Wiki: A collaborative environment for technical information related to AIX.

- Safari bookstore: Visit this e-reference library to find specific technical resources.
- developerWorks technical events and webcasts: Stay current with developerWorks technical events and webcasts.
- Podcasts: Tune in and catch up with IBM technical experts.

Get products and technologies

- IBM trial software: Build your next development project with software for download directly from developerWorks.

Discuss

- Participate in the developerWorks blogs and get involved in the developerWorks community.
- Participate in the AIX and UNIX forums:
  - AIX 5L -- technical forum
  - AIX for Developers Forum
  - Cluster Systems Management
  - IBM Support Assistant
  - Performance Tools -- technical
  - Virtualization -- technical
  - More AIX and UNIX forums

About the author

Martin Brown
Martin Brown has been a professional writer for more than seven years. He is the author of numerous books and articles across a range of topics. His expertise spans myriad development languages and platforms -- Perl, Python, Java™, JavaScript, Basic, Pascal, Modula-2, C, C++, Rebol, Gawk, Shellscript, Windows®, Solaris, Linux, BeOS, Mac OS X and more -- as well as Web programming, systems management, and integration. He is a Subject Matter Expert (SME) for Microsoft® and regular contributor to ServerWatch.com, LinuxToday.com, and IBM developerWorks. He is also a regular blogger at Computerworld, The Apple Blog, and other sites. You can contact him through his Web site.

Trademarks

IBM, AIX, and AIX 5L are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.
Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
Linux is a trademark of Linus Torvalds in the United States, other countries, or both.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
UNIX is a registered trademark of The Open Group in the United States and other countries.